Helsinki Metropolia University of Applied Sciences
Degree Programme in Information Technology

Anton Berezin

Content Management System for a Personally Driven Website

Bachelor's Thesis. 26 April 2009
Supervisor: Erkki Aalto, Principal Lecturer

Abstract

Author	Anton Berezin
Title	Content management system for a personally driven website
Number of Pages Date	40 26 April 2009
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Supervisor	Erkki Aalto, Principal Lecturer

The goal of the project was to develop a simple content management system that could be easily used both as a standalone web application and as a part of a more complex system. The application was supposed to consist of several key modules: commentary engine, search engine, navigation system and authentication system.

To create a content management system the implementations of the most common design patterns provided by PHP Zend Framework were used. The application was developed and tested on Ubuntu Linux with Apache web server and MySQL database server.

The final version of the project represented a content management system with a navigation system based on item categorization. The application consisted of multiple classes: the ones provided by Zend Framework and the ones created to fulfill application-specific requirements. The system extensively used the benefits of object oriented programming and the model-view-controller design pattern.

The results of the project allowed making several conclusions. First, although it is faster to develop the application using the agile programming model, such a technique leads to less optimized code then, if the design of the application was supported by a unified modeling language. Second, object-oriented programming provides the application with a high level of reusability and also makes it possible to employ only certain parts of the software.

Keywords	CMS, Zend Framework, MVC, PHP5, Design pattern

Contents

1 Introduction

The Internet has become ubiquitous in much of our society. Most corporations, nonprofit organizations, universities or schools have there own websites nowadays. There are many companies providing various Internet-based services. In order to compete successfully it is critical for them to be able to adopt the solutions implemented by the predecessors. There are several reasons to do so. First, avoiding reinventing the wheel lets bypass the most common errors assuming that somebody has already done and fixed those errors in the past. Second, reusing the solutions allows to save time and, as a result, economy money. The reusability concept is especially common in web-oriented applications because such applications normally share the same principles of data processing.

Currently a vast majority of websites or web applications are based on a certain type of a Content Management System (CMS). CMS is a computer application used to manipulate various types of data in order to generate content that is easily accessible by the end user. Data manipulation includes its creation, storage, modification and publication. Although there are many CMSs already existing, they require a programmer to get a certain amount of knowledge about it before he or she can easily operate with such an application.

The goal of my project is to provide a CMS that could be easily used both as a standalone web-application and as a component of a more complex system. The CMS consists of multiple modules sharing common object-oriented code. The usage of object-oriented programming paradigms lets easily extend the application by adding new modules or modifying the existing ones. The objective of the current report is to provide a comprehensive overview of industrial design patterns used in the project and the modules implemented to allow the CMS to have its basic functionality.

2 Theoretical background to Content Management Systems

2.1 CMS in General

CMS is a web application that normally allows its user to access the content stored on a particular server or even several server machines. In addition it is also expected to have certain administrative capabilities. For instance, a capacity to create new pages or alter the already existing ones might be a useful property to add to any CMS.

The most part of modern websites consist of a database to store data and the program logic to present such data in a human-readable form. In the past only companies, nonprofit organizations, universities or schools used to have their own websites. However, nowadays more and more people (or groups of people) tend to create pages for themselves. [1,1-2] In the majority of cases personal websites are wikis or blogs or probably a combination of both. The mentioned categories are briefly examined below.

2.1.1 Blog

Blog is a way to permanently store information and share it with the others in order to get a feedback. The majority of blogs are permanent and have the archive that allows the visitor to easily overview the posts written in the past. Additionally the blogs normally allow the owner to categorize or tag his/her posts so that the visitors are able to easily find what they want. The owner of the site normally gets a feedback in the form of comments to his/her posts. However, apart from standard commentary engine the blog might have a ranking system counting the amount of visitors or a rating system that allows visitors to grant a certain amount of points to the blog entity. Blogs are a remarkable part of the present World Wide Web.[2,XV-XVII] Figure 1 demonstrates the interface of a typical blog based on a popular CMS WordPress.

Figure 1: A typical blog

Figure 1 shows the user interface (UI) of a typical blog. There is a content area, a navigation area and a search engine's text box. The search engine is quite typical for any website nowadays. First, it lets find the content just by typing certain keywords in the text box. Second, if properly configured, it provides popular external search engines such as Google with a way to index the pages. As a result the potential of the site to attract more visitors increases.

2.1.2 Wiki

In contrast to blog, wiki is a free form website that consists of multiple editable web pages. The goal of such a CMS is to let anyone who accesses it to contribute or modify content, using a simplified markup language. Wiki concept is frequently used to power corporate or community pages. [3,9] Access to the pages is normally granted to a limited group of people related to a particular project or a company. However, there are exceptions. Public encyclopedia Wikipedia is an example of wiki-based site with the access given to all visitors. [4,27-29] Figure 2 provides an overview of wiki pages of Archlinux - one of Linux distributions.

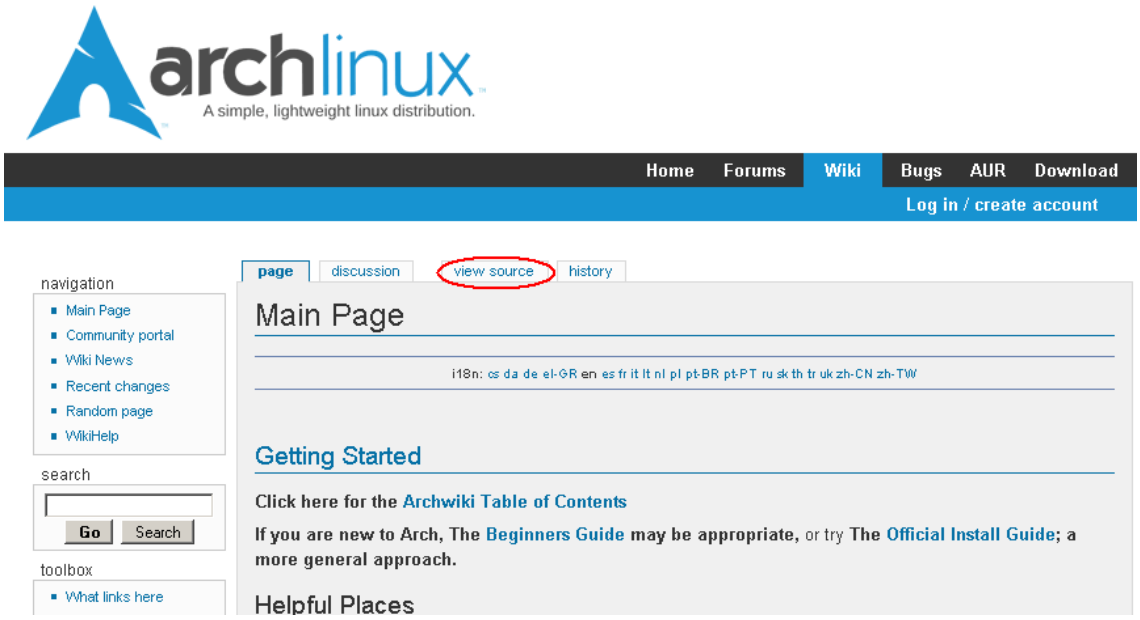

Figure 2: A typical wiki-based site

Figure 2 displays the UI of a common wiki page. As well as a blog such a site has a navigation area, a content area and a search engine's text box. However, additionally it allows authorized users to edit the content. Archlinux allows visitors to do it by clicking the link "view source" (highlighted by red oval in figure 2).

2.1.3 Multipurpose CMS

Combinations of blog and wiki approaches might become community websites, forums or even social networks. For instance, in a popular social network Facebook a user can register and log in, add some content such as notes or pictures and comment his/her friends' data. It is oblivious that although the social network is a composite web application, it shares the principles of content creation and manipulation with blogs and wikis.

The majority of companies nowadays have their own community pages to let the employees share the knowledge. Such sites normally do not have public registration in

order to assure that only the persons from a certain enterprise are able to access confidential information.

In addition to internal pages certain associations possess public forums, so that their customers are able to give feedback or communicate with each other. Figure 3 shows forum of one of the most popular Linux distributions - Ubuntu.

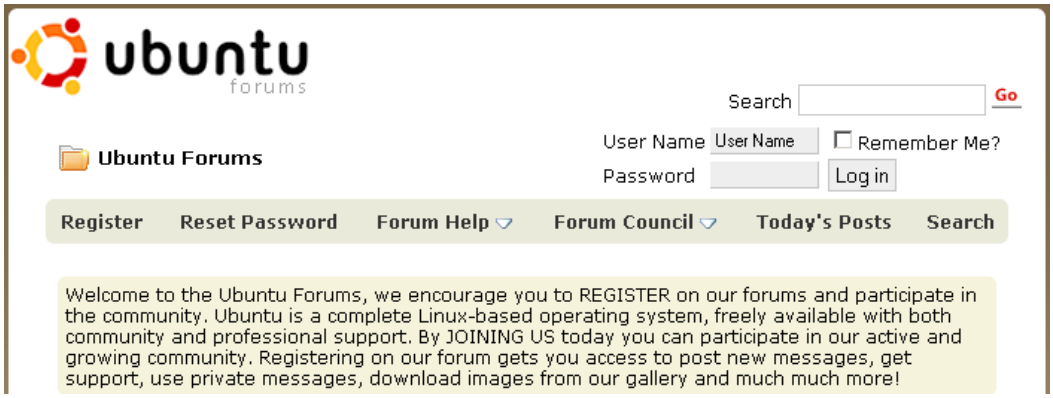

Figure 3: Ubuntu forum

Figure 3 illustrates the idea of public community. Once a visitor registers he/she is able to get help not only from other community members but from professional support team as well.

2.1.4 Common CMS Components

From programmers point of view any CMS (even the most sophisticated one) consists of a number of basic components to provide elementary functionality and some extensions to customize its behavior. The most common CMS components (or modules) are listed below.

To start with, the most important component of CMS is a content viewer. The most basic view is the one presenting textual metadata. Such a view can be called the core component of a CMS. There can be views that demonstrate a certain type of information such as

images or video objects. In addition to standard data it is a good idea to display copyright data as annotations. [1,264-265]

In any CMS there must be a way to log in for an administrator. Administrator is normally a person who takes care of the pages and the site on the whole. Administrator should be provided with instruments to create, modify and delete content. A simple JavaScript-based online text editor synchronized with the server can be a proper choice of tool combining all capabilities needed for content manipulation.

Both for the visitors and for the administrator a navigation system is required. Normally the content is assigned to miscellaneous categories and the system provides an easy way to browse such categories and move the pages from one category to another. On the client side navigation is represented by a certain type of menu that basically can be anything.

If the CMS is supposed to allow feedback it should have a commentary engine and authentication system to prevent unauthorized users or spam bots from commenting. Commentary engine is usually a simple text-form that passes its value to the server when submitted. Authentication, in contrast, is more sophisticated approach. There are two widespread techniques to track user's identity. Both of them are described below.

First, local authentication is employed in the majority of websites. If the user's credentials exist on the server he/she can log in. Otherwise he/she must register (if registration is open). A plain text file or a database can be an employed as user data storage place. Second, OpenID authentication method is frequently used nowadays. The key principle is that Internet companies with big amount of users such as Google or Yahoo! provide public methods to access their databases for user validation.

In the majority of cases the modules are slightly modified but in general keep their behavior as specified above. To be as extensible as possible the modules normally implement or use the implementations of multiple industrial design patterns such as Model View Controller

(MVC) and front controller.

2.2 Design Patterns in CMSs

Design pattern is a generic reusable solution of a common problem in the field of software engineering. It is an abstract description that can be adopted in different situations. The patterns are frequently employed in object oriented programming to simplify the code and ease its maintenance [4,XVIII-XIX]. The patterns most frequently used in web programming are examined below.

2.2.1 MVC

MVC is a design pattern successful use of which lets separate data from the way it (data) is presented to the user. In such a case the application is divided into model, view and controller. *Model* represents application's data. *View* is responsible for the way the data is shown to the observer. And *Controller* is in charge of processing the requests and making the adequate responses. [5] Figure 4 demonstrates the relationship between the model, the view, and the controller.

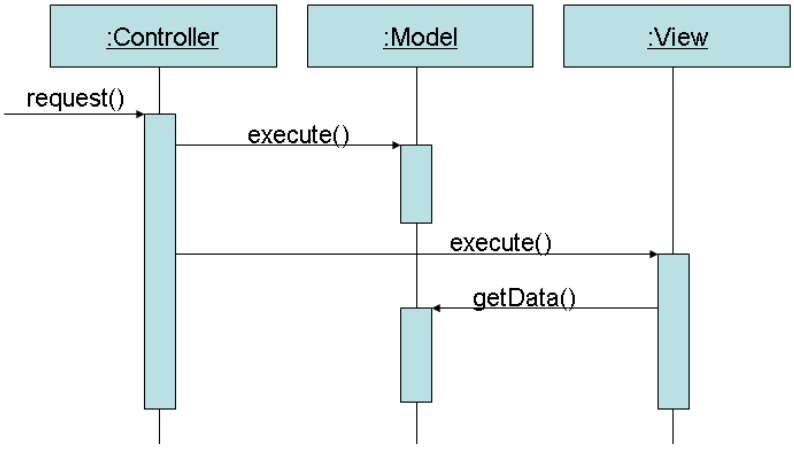

Figure 4: MVC: typical scenario

Information shown in figure 4 can be verbally stated as follows. When the model alters

application's data the controller notifies all the dependent views about the changes and the views perform the update operation by fetching new data from the application's model.
In practice, web applications use controllers to obtain data from the model and then pass it to the view. The view renders obtained data and displays it to the client. To assure that the pattern does what it is intended to do programmers do not recommend granting responsibilities of the model to the view and vice verse. The key point of this separation is the direction of dependencies: the presentation (or view) depends on the model but the model does not depend on the presentation. [6, 330-331] MVC is often used together with front controller design pattern. The front controller pattern is examined below.

2.2.2 Front Controller

The front controller consolidates all request handling by channeling requests through a single handler object. This object can carry out common behavior, which can be modified at runtime with decorators. The handler then dispatches to command objects for behavior particular to a request. [6, 344] Having the front controller allows programmers to avoid code duplication and as a result makes the application more configurable. Figure 5 shows a typical scenario of request handling using the front controller pattern.

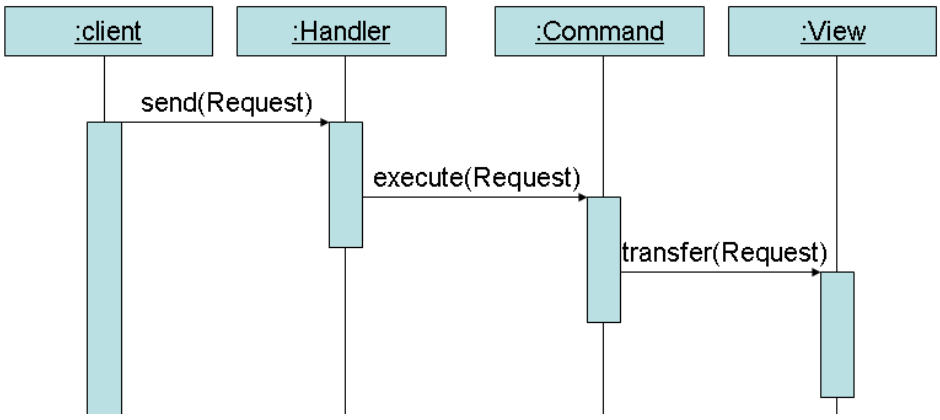

Figure 5: Front controller: typical scenario

In figure 5 command corresponds to a specific action (for example *indexAction()*). As soon

as the command's object completes its task the command decides which view to use to render the page in order to display it to the client.

The front controller is a more sophisticated pattern than its counterpart *page controller.* And its relative complexity results into a certain amount of advantages. First, only a single front controller has to be configured on the server. The handler does the rest of the dispatching. Second, due to the fact that the commands are dynamic, it is possible to add new actions without changing anything. [6, 346]

2.2.3 Page Controller

Page controller is an object that handles a request for a specific page or action on a website. It has one input controller for each logical page. The controller can be represented both by the page and by a separate object linked to that page. Practically the controllers are tied in to every action such as clicking a link or a button. [6, 333] Figure 6 demonstrates a correlation between the model, the view and the page controller.

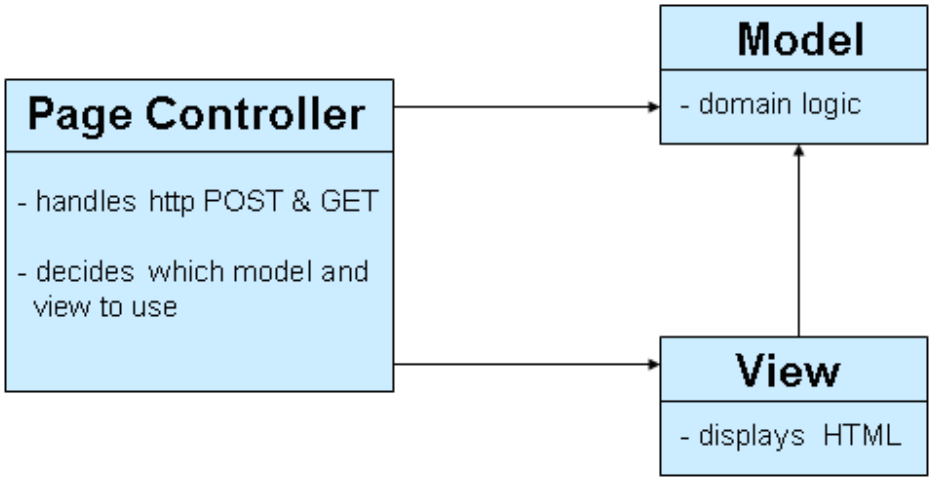

Figure 6: Page controller

In figure 6 it is shown that the page controller is an instance of MVC. The described design

pattern is easy to implement and it is frequently used with its counterpart the front controller. The popularity of such a bind is based on the simplicity of its configuration. The front controller is responsible for request redirection to various page controllers. The last mentioned ones represent the key application logic.

2.2.4 Active Record

The essence of active record design pattern is an object that wraps a row in a database table or a view, encapsulates the database access and adds domain logic on that data [6,160]. Usage of this pattern allows the programmer to avoid coping with basic database queries every time the table is created or modified. For instance, when utilizing the database directly, the code for inserting data might look the following way (assume that PHP language is used):

```
$db->query('INSERT INTO some_table VALUES (value1, value2, value3)');
```

Database command mentioned above is easy to write but absolutely impossible to maintain because if table's structure is changed the query has to be modified as well. The same operation using active record pattern's principle would look like the code below:

```
$some_table->column_name1 = value1;

$some_table->column_name2 = value2;

$some_table->column_name3 = value3;

$some_table->save;
```

Although such a code consists of more lines then previously cited one, it is totally independent from the table structure and has to be changed only if column names are changed.

Total independence of the code from the database structure allows the programmer to add

new features on the fly without revising the whole application. Because of such a flexibility provided by the usage of active record pattern, the majority of currently existing CMSs and frameworks employ it. For example, Ruby on Rails and Symphony frameworks contain the implementations of the pattern described above.

2.3 Data Indexing and Searching

For any website it is critical to provide a user with the ability to find any data depending on what he/she wants. The navigation systems based on menus, tags or sections have static data categorization. The word *static* in such cases means that all the categories are predefined and links to the categories are saved permanently on a server machine. The key disadvantage of employing only static categorization is that any user has to go through all the links manually in order to find some information. To help the user with finding data the majority of websites employ various search engines.

Search engine is a tool designed to help with search the data. Data basically can be anything: text, document, image or video. The data can be located on a local machine or in World Wide Web (WWW). In order to allow searching data must be indexed. The most famous Internet search engines such as Google and Yahoo! use so-called crawlers in order to find and index new information. Crawler is a program that automatically and methodically browses the WWW and fetches up-to-date data. When the page is found it is stored locally on a server. When somebody tries to find information a query to the search engine is sent. As soon as the engine gets the query, it goes thorough the indexed data and returns an appropriate response. [7, 6-8; 7, 66-68]

In contrast to big search engines local ones may avoid using crawlers by indexing the data just when it is added to the server. The purpose of having local search is to let the visitor find information related only to a particular site. A good choice of local search engine is Lucene. Lucene is open source software originally implemented in Java. The engine is totally independent from the way data is stored on the server because it keeps its own index

on the local machine. Different types of data besides HTML pages. For instance, PDF and DOCX files can be indexed by Lucene as well. [8, 76]

There are two key components in the Lucene search engine: *IndexWriter* and *QueryParser*. The first one is responsible for serializing data into a specific format so that data can be found whenever needed. The second one has to analyze the incoming queries and look through the index in order to return a sufficient result. [8, 77] Nowadays Lucene is ported on many different platforms besides Java such as PHP, C++, C#, Python, Ruby, Perl and Delphi.

3 Implementation of a CMS

3.1 Project Limitations

Before the actual coding process I had to set certain limitations to the project, not to delay its development for too long. The initial plan was to create a CMS that would consist of the following modules: authentication and registration module, a commentary engine, an admin panel, a category based navigation and a search engine. Eventually it was expected to get an application that would be comfortable to use without studying the whole code.

In order to make the CMS reusable it was critical to apply the design described in the literature overview. However, implementing those patterns from scratch was totally against the goal of the project because any new implementations of the patterns are to be examined before they can be successfully employed. To guarantee that the CMS working principle would be clear for any other developer but me, I decided to use well-known technologies and industrial standards. The choice of the technology is reviewed below.

3.2 Preliminary Choices

3.2.1 Platform

According to *LAMP: The Open Source Web Platform* by Dale Dougherty one of the most popular bundles for developing web applications is Linux, Apache, MySQL and PHP (LAMP) [9]. Selection of PHP as a server-side programming language guaranteed that it would be possible to get support from a large community of PHP programmers if any trouble occurs. As a result I was able to solve any code-related problem in a relatively short period of time.

To host the LAMP bundle I selected Ubuntu Linux distribution because it provides an easy to use installer and is based on stable Debian Linux. The key feature of Debian is the

Aptitude package manager that allows the user to obtain new software by typing a single line in the terminal:

```
sudo apt-get install NAME_OF_APPLICATION
```

In order to be installed, the application must exist in the distribution's online package database.

3.2.2 Framework

Before I actually started developing my CMS, I decided to choose a framework in order to write a standardized and highly reusable and configurable application. At that time there were several leading solutions on the market. Below three of the most popular ones are examined.

First, Symfony framework that aims to speed up the creation and maintenance of web applications, and to replace the repetitive coding tasks by various code generators. Symfony has been used worldwide in a number of enterprise-level applications, most notably by Askeet and Yahoo! Bookmarks. The key feature of the framework is the use of Command Line Interface (CLI) and YAML – a markup language used to store configurations in Symfony. [10]

Second, Cake PHP with the majority of its concepts borrowed from Ruby on Rails. It aims to bring simplicity and scalability to PHP frameworks. It does not depend on miscellaneous configuration files as much as Symfony, instead it motivates the developer to follow certain coding conventions in order to make his/her application work properly.

Third one to be mentioned is a use-at-will Zend Framework (ZF). A use-at-will framework is the one that consists of loosely coupled classes that can be used independently from each other. Loosely coupling describes an approach where the interfaces are developed with minimal amount of dependencies between the parties and it guarantees that if a particular

module is changed the others remain unaffected. ZF contains implementations of the MVC, Table Data Gateway, and Row Data Gateway design patterns. Zend Framework provides individual components for many other common requirements in web application development. [11] Instead of having strict coding conventions or various configuration files ZF has reasonable defaults that can be overridden according to application's specific requirements.

Due to the fact that the goal of the project was to use industrial standards (not the ones provided by the framework), the selection was easy. Zend Framework was chosen to be the backbone of my CMS because it does not make the developer follow any non-conventional coding practices.

3.3 Installation and Configuration

3.3.1 Environment

The first step was to install Apache web server. This was done by typing the following in the terminal:

```
sudo apt-get install apache2
```

After the installation the server was tested by typing in browser's address bar the line:

```
http://localhost
```

As a consequence a folder entitled "apache2-default/" appeared in the browser. When the folder was opened a message "It works!" was displayed.

The second step was to install PHP 5. They key difference between fifth version of PHP and all the previous ones is that PHP 5 can be used for object oriented programming meanwhile the other versions cannot. In order to set up and bind PHP 5 with Apache server

the following command was used:

```
sudo apt-get install php5 libapache2-mod-php5
```

After the installation, the Apache server had to be restarted:

```
sudo /etc/init.d/apache2 restart
```

By default in Ubuntu “/var/www” folder is used to store data accessible over internet. To test PHP configuration I created “test.php” page with the following code inside:

```
<?php phpinfo(); ?>
```

Afterwards the page was accessed in the browser at the following address:

```
http://localhost/test.php
```

As a result the browser displayed information about the current state of PHP.

MySQL server was installed using the same tool as the one employed for setting up Apache and PHP.

```
sudo apt-get install mysql-server
```

To make the database server secure setting a proper password was required. It was achieved using the terminal:

```
mysql –u root
```

```
SET PASSWORD FOR 'root'@'localhost' = PASSWORD('yourpassword');
```

The last step was to set up a program called phpMyAdmin which is a popular tool to administrate databases. The terminal was used again:

```
sudo apt-get install libapache2-mod-auth-mysql php5-mysql phpmyadmin
```

To bind MySQL with Apache, I had to modify "/etc/php5/apache2/php.ini" file. I uncommented the line (removed the semicolon at the beginning of the line):

```
extension=mysql.so
```

In the end the web server had to be rebooted again:

```
sudo /etc/init.d/apache2 restart
```

3.3.2 General Structure of the Application

According to Zend Frameowork's tutorial, the default structure of the application had to consist of three major parts: the framework library itself, application specific classes, configurations, templates and publically accessible content such as JavaScript files, CSS style sheets and images. [12] The directory structure I used in my CMS is shown in figure 7.

Figure 7: Project directory structure

As shown in figure 7 the “library” folder contains all the classes provided by Zend Framework. A folder labeled “public” has publicly accessible content. Inside the “application” folder there are several subdirectories. All application specific classes and models are inside the directory named “classes”. The “layouts” directory stores generic templates that are used to render any page generated by the application. The layouts are used as decorators for the views stored inside the directory with a respective name. MVC controller’s implementations are kept in the “controllers” folder. To draw the line, all application configuration files are placed inside the “config” folder.

3.3.3 Request Rewriting

In order to let the web application have only a single front controller that would handle and dispatch all the incoming requests, I had to create a proper “.htaccess” file in the website’s public directory. The “.htaccess” file is intended to contain folder specific server directives. In my case, for example, the goal was to redirect all the requests to “index.php” script if they are not related to the files inside the public directory. Such files might be images, JavaScript scripts, style sheets or any other documents. My “.htaccess” file had the following content:

```
RewriteEngine on

RewriteBase /

RewriteRule !\.(js|ico|txt|gif|jpg|png|css)$ index.php
```

The first line states that request rewriting is enabled. The second one says that the rewriting rule affects only the parent directory of “.htaccess” file. The last line is a directive to pass the request to “index.php” script if it is not related to a file with extensions listed in the brackets.

3.4 Application Design

3.4.1 Data Model

The first step of application design was planning the data model of the CMS. The model describes miscellaneous database tables and PHP classes that wrap the tables. For the commentary engine two tables were required: the one to store users' credentials and the one to keep actual comments. There had to be separate tables to store pages and categories as well. The Resulting database design is shown in figure 8.

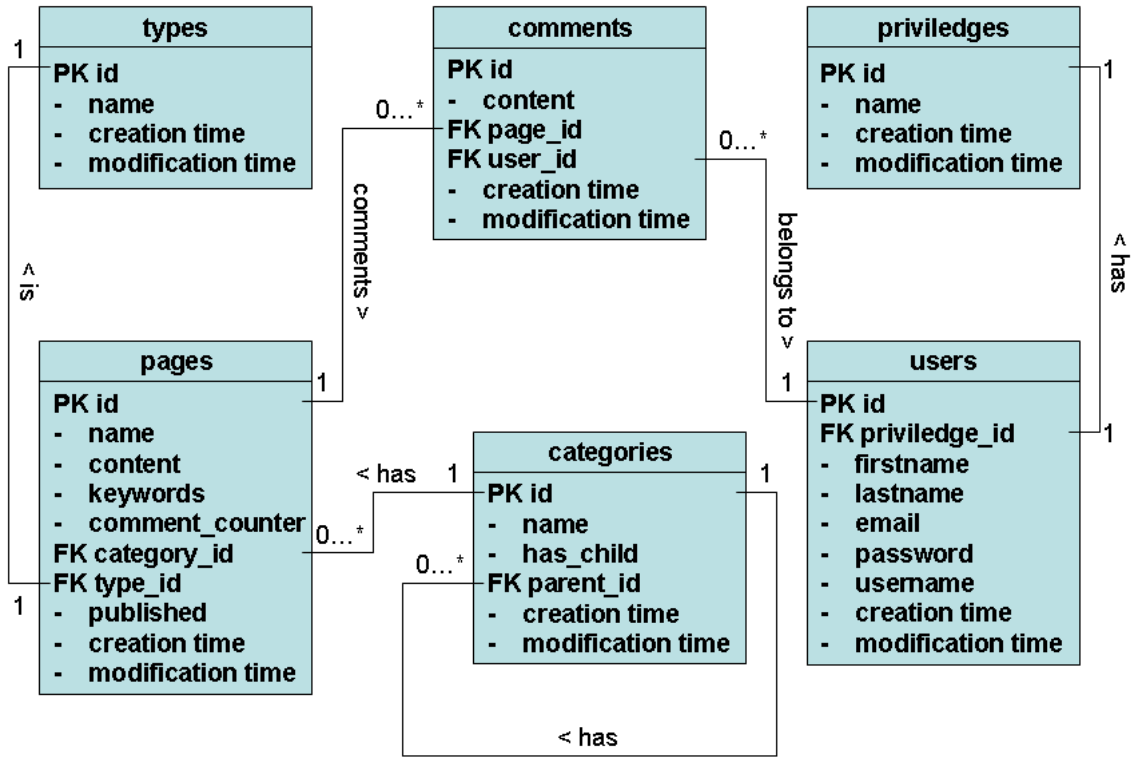

Figure 8: Database model

In figure 8 there are two tables that were not specified above. First, the table called "types". It simply describes what kind of content the page contains. The table is used by the application to dynamically decide how to render the page. For example, if the content is an

html text, it is directly included into the page's body. However, if the content is an image, it is used to generate an image tag to be included into the resultant page. Second, the table called "privileges". It specifies what type of operations the user is able to perform when he/she logged in. For instance, the administrator is able to observe, create and modify data, and meanwhile a visitor can only observe and comment the pages.

In addition to the extra tables mentioned above, there is one more issue to be explained – structure of the table called "categories". As it can be deduced from the table's name, it is responsible for storing categories. Any category can contain zero or more subcategories, and this is why the table has 1 to 0…* relation to itself.

I developed exactly six classes to provide an interface for the database. Each of them had the same name as the table it was responsible for. For example, the class that correlated with the "comments" table was named "Table_Comments". In addition to model classes I created four form ones. Form classes were designed to let the user add/modify data in the tables. "Types" and "privileges" tables did not have their own forms. I made a decision to do so because the data that the two tables possessed was application-specific and was supposed to be hardcoded in advance.

3.4.2 Controllers

In order to make coding simpler I separated the application's logic into several independent controllers. From the functional point of view there was no difference between having a single controller with a variety of actions and having several controllers with just a few. However, from the developer's point of view there was a difference. If there was a single controller, the programmer would have to modify the existing class when a new feature was added. At the same time, multiple controllers provided a possibility to leave the existing classes untouched by creating a new controller in a separate file whenever required.

As recommended in the Zend Framework guide book, I copied the error controller mentioned in the tutorial in order to track all the errors that could occur in my application. The key objective of an error controller is to catch the exceptions whenever they are thrown and display the messages in a web-browser. [13]

The first controller I developed was an "authentication" one. In the beginning it was responsible for three actions: registering a new user, logging in and logging out. Later on three more actions were added: editing the user's data, showing all the users and deleting a particular account. The controller mentioned above was developed first because the first event supposed to be happening with the user when he/she entered the site was "authentication".

Later on I created three more controllers: "category", "page" and "comment" ones. They actually had four common actions: "show", "add", "edit" and "delete". Afterwards I wrote an additional "image" controller to support image uploading. Due to the fact that all of the controllers and their actions were completely independent from each other their unit testing was done just by typing addresses in the web-browser as follows:

http://domain_name/controller_name/action_name?param=value

After all the key controllers were tested and fixed I implemented the index controller. The index controller is the one that responds to the requests when a user types only web resource's name in his/her web-browser's address bar. In my CMS the controller had to check if the user logged in and to redirect him/her to the authentication page on failure. If authentication was successful, the controller had to provide the user with access to the main application's logic.

To provide a generic interface to all the controllers, Zend Framework has a special basic class called "Zend_Controller_Action". Although it is possible to write all the controllers as extensions (child classes of) to the "Zend_Controller_Action", it is a good idea to have an intermediate class to contain common application's logic such as authentication function calls or checking user's privileges. According to Vacca J R PHP5 has multiple vulnerabilities that may let a hacker get control of the application [14, 157]. To ensure that SQL injection or remote code execution would never occur, I placed the code checking all necessary parameters inside the *init()* function of the intermediate class mentioned above. In Zend Framework *init()* function is automatically executed before the controller performs any data manipulation task. Encapsulation of all the security-related logic within a single class allowed me to write a cleaner code by concentrating on task-specific functionality of a particular module.

3.4.3 Authentication and Registration

In my CMS I decided to have an open table-based authentication. In order to properly employ such a one, I used a class provided by ZF. The following code was responsible for connection of the authentication system with the database.

```
$db = Zend_Db::factory($config->db);
$auth = new Zend_Auth_Adapter_DbTable($db, 'users', 'username',
'password');
```

As it can be seen above in Zend Framework, the class responsible for table-based

authentication is called *Zend_Auth_Adapter_DbTable.* Several parameters are passed to its constructor. First, *$db* is an object that represents the connection with the database. The connection with the database is created using the information retrieved from the configuration file. Second, the table name ("users") and the names of the columns storing *username* and *password*.

Due to the fact that just a single table was used for authentication, the development of the registration was simple. In order to add a new user, he or she had to fill in the form shown in figure 9.

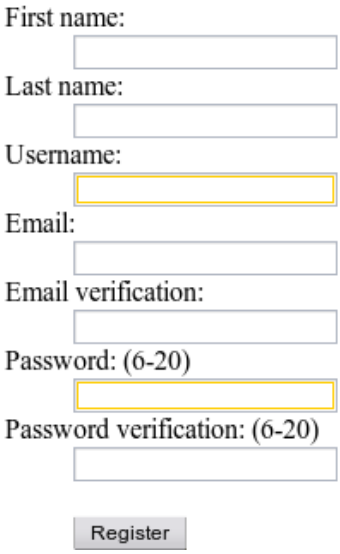

Figure 9: Registration form

Registration form shown in figure 9 contains the fields that are common in such type of forms. Before a visitor registers all data is validated by the form class. An error will be displayed if the user already exists in the database or if the information typed into the form does not pass the validation.

The process of form validation was designed to be straightforward. First, the data was obtained from the request (POST one in my case). Second, the data was passed to the form in order to be validated. Technically it was implemented the following way:

```
$request = $this->getRequest();
$form = new Form_Register('/authentication/register');
if ($this->getRequest()->isPost() && $form->isValid($request->getPost()))
        {
        //insertion into the database
        }
```

If the validation succeeded the data from the form was inserted into the database.

Actually, all the forms created using ZF are processed using a similar technique according to the recommendations given by ZF documentation. That is why the code written looks clear and is easy to maintain.

3.4.4 Image Uploading and Thumbnail Generation

The process of adding normal pages to the site was based on filling in the standard form. If the form was validated, new entity was added to the database. The process of filling in the form and validating it coincides with the registration form validation – overviewed in section 3.4.3.

Besides adding HTML based pages I wanted to provide the administrator with a way of adding images, so that they were displayed as if they were normal pages. To do so I created a custom view for the database entities describing path information of certain images. As it was shown in figure 8 every page had a column that identified its type. In my case there were only two types: a pure HTML page and an image. Whenever the application was rendering the view, it was detecting the type and dynamically choosing the appropriate way of presenting data. For example, when the content field contained normal text, the text was included straight into the page, meanwhile if the content field described image's name, the view script used that name to create a proper <*img*> tag.

In order to make image uploading possible I created a custom form with a file upload

capability. Instead of writing any content manually the user had to pick an image from the hard drive (or an alternative to it) and fill in the description of the image: name and keywords for search engine. When the form was submitted it was given a unique name and uploaded to the "/public/images" directory. The unique name was generated by the function written below:

```
public static function randStr($length)
        {
        $code = md5(uniqid(rand(), true));
        if ($length != "")  { return substr($code, 0, $length); }
        else  { return $code;}
        }
```

In order to generate a random string I employed standard random number generation and md5 encryption. The return value was a string consisting of mixture of digits and letters.

When observing normal pages inside a category, nothing else but the name of the page and keywords related to it were supposed to be displayed. To make observing the images simpler for the end user I decided to additionally demonstrate thumbnails of the images while looking through the categories. Due to the fact that Zend Framework did not have any image processing classes I had to employ a third party solution.

One of the best approaches to operate with images using PHP5 I managed to find was *thumbnail.inc.php* class. It supported the majority of key image-processing functions such as rotating, cropping, resizing to name but a few. For actual thumbnail generation I planned to use only the resizing function.

The actual utility function I employed had the following structure.

```
public static function make($sourcePath, $destPath, $w, $h, $q = 100)
        {
        $thumb = new Zend_Image_Thumbnail($sourcePath);
```

```
$thumb->resize($w, $h);
$thumb->save($destPath, $q);
}
```

In the function the parameters can be explained as follows. $sourcePath describes full name (path and the actual filename) of the image stored on the server. $destPath is related to the full name of a thumbnail to be generated. $w and $h stand for width and height respectively. $q specifies quality of the thumbnail in percents relatively to the original.

In the controller I invoked the function the following way:

```
Utility_Thumb::make(PUBLIC_PATH."/images/".$newName,
PUBLIC_PATH."/images/_".$newName, 150, 150);
```

When such a command was executed a new thumbnail was created. It had maximum width/height of 150 pixels and its name was actually a name of the original with an underscore in the beginning.

3.4.5 Data Indexing and Search Engine

In order to make local search possible I decided to index the pages whenever they were added to the database. The following commands were supposed to be invoked when valid data was added to the ‘pages’ table:

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('keywords', $keywords));
$doc->addField(Zend_Search_Lucene_Field::UnStored('content',
$data['content']));
$doc->addField(Zend_Search_Lucene_Field::Text('name', $name));
```

```
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('did', $id)); //did =
id of the item in database
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('type', $type));
```

As it can be seen from the code written above, the search index was supposed to contain the same information as the database but in a format accessible by Lucene. I wrote the following lines of code in order to remove the page out from the search engine's index whenever the row is removed from the database table.

```
$hits = $index->find('did:' . $where);
foreach ($hits as $hit) {$index->delete($hit->id);}
```

The search engine finds an entity inside the index that is related to the page being removed from the table and deletes the entity.

Whenever the page is updated, the data is first removed from the index and later on is added there again. Such a way of maintaining the indexed data is required because the Lucene does not have an *update* function – only *insert* and *delete* ones.

4 Results

4.1 Basic Requirements

In order to work the CMS has to be provided with a certain environment. First, the environment has to support PHP5. Second, there must be some database server. Although I used MySQL when developing the CMS it is really easy to replace MySQL with any other relational database server. Third, there must be a web server to get requests and send responses. From the information stated above it is clear that the only *strict* requirement for my web application to be employed successfully is the usage of PHP5. Any other components can be of a custom choice.

Apart from the requirements listed previously a public directory to store uploaded files must have read and write permissions for everybody: owner, group and the others. Additionally, the root public directory has to have ".htaccess" file or an alternative to it for proper request redirection.

4.2 Functionality

The final version of my CMS consisted of the following modules: a commentary engine, a navigation/categorization module, a search engine and an authentication system. Any registered user was allowed to look through the categories, observe the pages and comment them. However, only the administrator was granted a capability to create/edit/remove the pages. Registration was designed to be open.

The administrator was given the ability to modify content of the site and move the categories/pages from one place to another. In addition to this the administrator was allowed to observe the information about the users and delete the users. The interface to manipulate all types of data within my application is shown in figure 10.

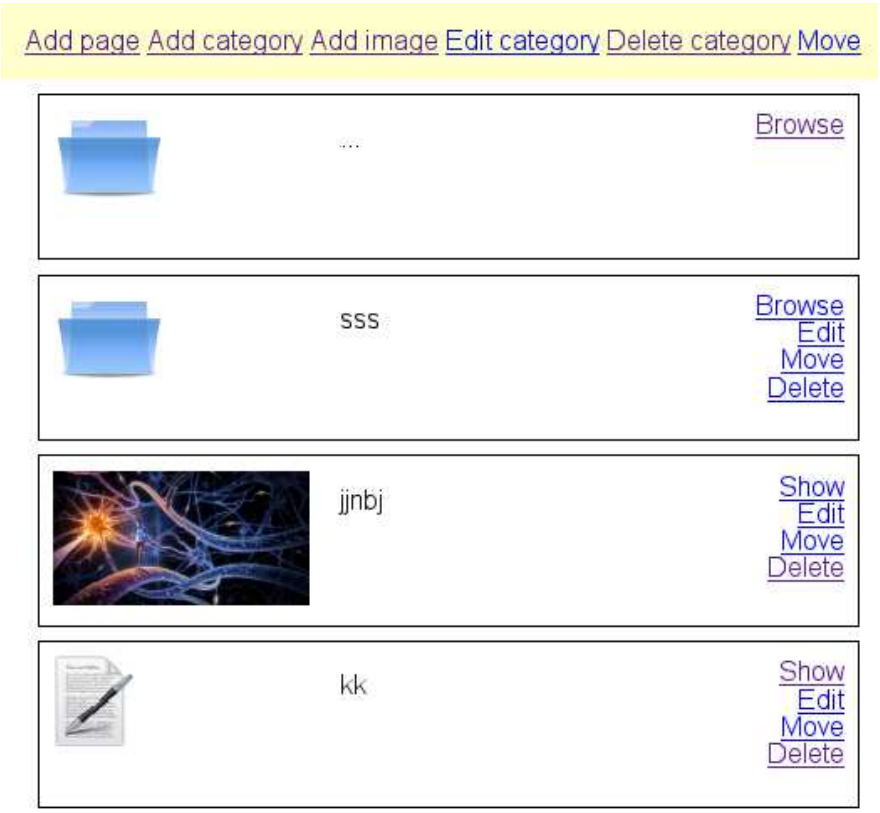

Figure 10: Core administration interface

In figure 10 three items of totally different types of data can be seen: category, image and standard page. All of them share common methods provided by *GenericController* class. The links located within the boxes manage with a particular item. At the same time the one in the top of the figure are related to the category currently being observed.

Searching any data was designed to be easy to use. In order to find something the pattern had to be typed inside a search text field. Afterwards the “Go!” button had to be pressed. The basic search interface and the results of a sample query are shown in figure 11.

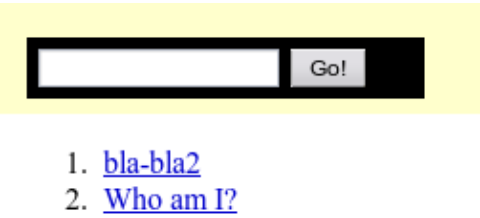

Figure 11: Searching

In figure 11 it is possible to see the results of a sample search query (keyword “bla” was typed inside the text box). The results are displayed as a numerated list of titles of the pages linked to the pages.

4.3 Reusability and Customization

Due to the fact that MVC design pattern was used in my CMS, it was extremely easy to extend its functionality by adding new controllers/views/models. Additionally the core components of my application provided Asynchronous Javascript And XML (AJAX) support. If the developer wished to retrieve pure data from the server he/she had to add “ajax=true” GET parameter. The response from the server in such case would contain no decorative elements such as footer or header.

In order to provide easy to alter configurations I used Zend Config. This class was intended to read core application’s settings such as database parameters from the “*config.ini*” file located inside “*/application/config*” directory. All path constants were stored inside “*bootstrap.php*” file.

4.4 Errors and Problems

During the development process I encountered some problems and had to fix several errors. Below all of them are listed and examined.

First, when I was configuring image uploading I had a problem with folder permissions. To solve it I changed the mode of the directory containing the images to 777. This minor fix allowed my PHP scripts to upload the images from certain users without any obstacles. Second, in order to make image thumbnailing work I had to setup a PHP GD library. The library had the majority of basic functions required for image processing.

Third, I had to solve a problem of removing the categories properly. If I had only a single valid type of content in my CMS – HTML pages, the process of removal would be easily configured by adding “ON DELETE cascade” to every foreign key of each database table. However, due to the fact that apart from standard pages my CMS was supposed to support image uploading such a solution was not a proper one. The database engine could not remove the images themselves because they were stored in a file system. Only the references to the images could be removed. In such a case the server would have redundant files that could not be observed nor removed via category browsing in the CMS. The solution was simple – it was made impossible to remove the category if it contained any items inside.

5 Discussion

5.1 Benefits

The final version of my CMS totally fulfills the requirements I set when I started working with my project. I managed to develop a web application that can manipulate the categories and the items within the application as if they were real directories and files. The CMS has a potential to be extended because it was developed using object-oriented PHP5 and Zend Framework. Due to the fact that my application employs the MVC, it is possible to add new features without modifying the original source code.

On one hand, my CMS can be used as a standalone publishing tool. It provides open registration and capabilities to cope with different types of content. On the other hand, it can be easily extended and customized to serve as a part of a more complex system. For example, to make a wiki site, the developer has to grant all registered users an ability to create or edit content and modify the CSS files to make the pages look proper.

Since the application consists of multiple classes, the solutions I implemented, for example, for authentication, can be reused separately without employing the whole CMS. Moreover, it is possible to completely change the behavior of my application by just modifying the views and the controllers without touching the model. Additionally, if the database structure is to be changed, the model can be slightly modified to suit a new table structure. In such a case, if the model interface (function definitions) remains untouched, the application controllers and views might not be modified either. To draw the line, the usage of the MVC makes data totally independent of the way it is presented to the user.

5.2 Drawbacks

Although the usage of Zend Framework with the implementations of the majority of industrial design patterns makes the development process fast and the application easy to

maintain, it brings several drawbacks as well. First, Zend Framework is not created for small web applications such as personal blogs because it has an overhead caused by the implementation of the front controller. For example, to display a “Hello World!” message in browser’s window the controller has to be instantiated and later on the dispatch method has to be invoked. Without Zend Framework such a simple event could be done via just a single statement *echo ‘Hello World!’*. Another drawback is actually based on the first one. Zend-powered web applications are not for shared servers. The shared servers contain multiple websites that can be accessed by different users simultaneously and independently. Instead, the applications based on Zend should be hosted on their own dedicated web server.

The application is based on the usage of constants (global variables) describing the path to the most significant folders in my CMS. In object-oriented programming all the variables are supposed to be local and their values are to be passed via messaging between the object. In contrast with the local ones, global variables use shared memory and therefore their usage totally contradicts with the paradigms of object-oriented programming.

6 Conclusion

The goal of my project was to provide a CMS that could be easily used both as a standalone web-application and as a component of a more complex system. The CMS consisted of multiple modules sharing common object-oriented code. The usage of object-oriented programming paradigms allowed to easily extend the application by adding new modules or modifying the existing ones. The final version of application consisted of several modules: a navigation system based on item categorization, a commentary engine, a search engine and an authentication system.

To make the development process easy and fast, I used agile programming methodology. Agile programming means that the coding is started as soon as possible without making the UML diagrams. Such a technique allowed me to achieve the goal in a short period of time. However, instead of being optimized, my code had a certain number of duplicated patterns. If I had to remake my project, I would start from creating the UML diagrams. Additionally I would develop a data access system similar to the one used in the UNIX based operating system. I would add a separate table for all the events that could occur in the application. The table would have the relations with the existing *"privileges"* one.

To make the applications work faster, it would be a good idea to implement the most common design pattern in a lower level (in C language, for instance) to enhance the speed of the scripts' execution. Such an approach would lead to the creation of cleaner and more standardized code.

References

1 Mauthe A, Thomas P. Professional Content Management Systems. Hoboken, NJ: John Wiley and Sons; 2004

2 Fielix L, Stolarz D. The hands-on guide to video blogging and podcasting. Burlington, MA: Focal Press; 2006.

3 Ebersbach A, Glaser M, Heigl R, Warta A. Wiki: Web Collaboration. 2nd edition. Berlin: Springer; 2008

4 Shalloway A. Design patterns explained : a new perspective on object-oriented design. Boston, MA: Addison-Wesley; 2002.

5 Fowler M. GUI architectures [online]. ThoughtWorks; 18 July 2006.
URL: http://www.martinfowler.com/eaaDev/uiArchs.html
Accessed: 18 January 2009

6 Fowler M. Patterns of Enterprise Application Architecture. Boston, MA: Addison-Wesley; 2003.

7 Levene M. An introduction to search engines and web navigation. Boston, MA: Addison Wesley; 2005

8 Eubanks B D. Wicked cool Java. San Francisco, CA: No Starch Press; 2005

9 Dougherty D. LAMP: The Open Source Web Platform [online]. O'Reilly Media; 26 January 2001
URL: http://www.onlamp.com/pub/a/onlamp/2001/01/25/lamp.html
Accessed: 21 January 2009

10 O'Brien D. PHP frameworks [online]. IBM; 9 October 2007
URL: http://www.ibm.com/developerworks/opensource/library/os-php-fwk1
Accessed: 23 January 2009

11 Chase N. Understanding the Zend Framework [online]. IBM; 27 June 2006
URL: http://www.ibm.com/developerworks/opensource/library/os-php-zend1/
Accessed: 27 January 2009

12 Zend Framework Quick Start [online]. Zend; 19 January 2009
URL: http://framework.zend.com/docs/quickstart/set-up-the-project-structure
Accessed: 2 February 2009

13 Zend Framework Quick Start [online]. Zend; 19 January 2009
URL: http://framework.zend.com/docs/quickstart/create-an-error-controller-and-view
Accessed: 3 February 2009

14 Vacca J R. Practical internet security. New York, NY: Springer; 2007.